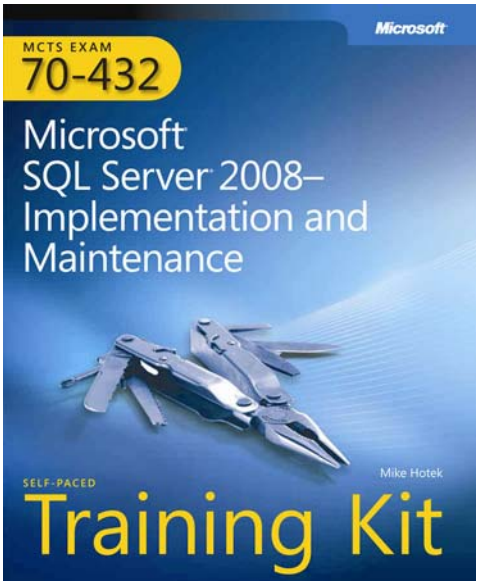

MCTS Self-Paced Training Kit (Exam 70-432): Microsoft® SQL Server® 2008—Implementation and Maintenance

Mike Hotek

To learn more about this book, visit Microsoft Learning at http://www.microsoft.com/MSPress/books/12858.aspx

9780735626058

Contents

CHAPTER 2

Database Configuration and Maintenance

The configuration choices that you make for a database affect its performance, scalability, and management. In this chapter, you learn how to design the file and filegroup storage structures underneath a database. You learn how to configure database options and recovery models. You will also learn how check and manage the integrity of a database.

Exam objectives in this chapter:

- Back up databases.
- Manage and configure databases.
- Maintain database integrity.
- Manage collations.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have:

- Microsoft SQL Server 2008 installed
- The *AdventureWorks* database installed within the instance

REAL WORLD

Michael Hotek

I have worked on millions of databases across thousands of customers during the portion of my career where I have worked with SQL Server. In all that time, I have come up with many best practices while at the same time creating many arguments among the "purists." All my recommendations and approaches to architecting and managing SQL Servers come from a pragmatic, real-world perspective that, although rooted in a deep knowledge of SQL Server, hardware, networking, and many other components, rarely matches up with the perfect world theory.

Designing the disk structures that underlie a database is one of the cases where I deviate from a lot of the theoretical processes and computations that you will find published. Although you can find entire white papers and even sections of training classes devoted to teaching you how to calculate disk transfer and random vs. sequential writes, I have never encountered an environment where I had the time or luxury to run those calculations prior to implementing a system.

It is really nice that there are formulas to calculate the disk transfer of a given disk configuration, and you can also apply statistical methods to further refine those calculations based on the random vs. sequential I/O of a system. However, all the time spent doing the calculations is worthless unless you also know the required read and write capacity of the databases you are going to place on that disk subsystem. Additionally, unless you are buying a new storage system, dedicated to a specific application, you will have a very difficult time architecting the disk storage underneath a database according to all the theories.

The challenge in achieving optimal performance is to separate the transaction logs from data files so that you can isolate disk I/O. The transaction log is the key to high-performance write operations, because the maximum transaction rate is bound by the write capacity to the transaction log file. After taking care of the transaction log, you need to add enough files and filegroups to achieve enough disk throughput to handle the read/write activity. However, the most important component of performance is to write applications with efficient code that accesses only the minimum amount of data necessary to accomplish the business task.

Lesson 1: Configuring Files and Filegroups

Data within a database is stored on disk in one or more data files. Prior to being written to the data file(s), every transaction is written to a transaction log file. In this lesson, you learn how to design the data files underneath a database, group the files into filegroups to link physical storage into a database, and manage the transaction log. You also learn how to configure the *tempdb* database for optimal performance.

After this lesson, you will be able to:

- Create filegroups
- Add files to filegroups
- Work with *FILESTREAM* data
- Configure the transaction log

Estimated lesson time: 20 minutes

Files and Filegroups

Although storing all your data in memory would provide extremely fast access, you would lose everything after the machine was shut down. To protect your data, it has to be persisted to disk. Underneath each database is one or more files for persisting your data.

SQL Server uses two different types of files—data and transaction log files. Data files are responsible for the long-term storage of all the data within a database. Transaction log files, discussed in more detail later in this lesson, are responsible for storing all the transactions that are executed against a database.

Instead of defining the storage of objects directly to a data file, SQL Server provides an abstraction layer for more flexibility called a *filegroup*. Filegroups are a logical structure, defined within a database, that map a database and the objects contained within a database, to the data files on disk. Filegroups can contain more than one data file.

All objects that contain data, tables, indexes, and indexed views have an ON clause that you can use to specify when you create an object that allows you to specify the filegroup where SQL Server stores the object. As data is written to the objects, SQL Server uses the filegroup definition to determine on which file(s) it should store the data.

At the time that a file is added to a database, you specify the initial size of the file. You can also specify a maximum size for the file, as well as whether SQL Server automatically increases the size of the file when it is full of data. If you specify automatic growth, you can specify whether the file size increases based on a percentage of the current size or whether the file size increases at a fixed amount that you define.

Unless a filegroup has only a single file, you do not know in which file a specific row of data is stored. When writing to files, SQL Server uses a proportional fill algorithm. The proportional fill algorithm is designed to ensure that all files within a filegroup reach the maximum defined capacity at the same time. For example, if you had a data file that was 10 gigabytes (GB) and a data file that was 1 GB, SQL Server writes ten rows to the 10 GB file for every one row that is written to the 1 GB file.

The proportional fill algorithm is designed to allow a resize operation to occur at a filegroup level. In other words, all files within a filegroup expand at the same time.

File Extensions

SQL Server uses three file extensions: .mdf, .ndf, and .ldf. Unfortunately, many people have placed a lot of emphasis and meaning on these three extensions, where no meaning was ever intended. Just like Microsoft Office Word documents have a .doc or .docx extension, and Microsoft Office Excel files have an .xls or .xlsx extension, the extension is nothing more than a naming convention. I could just as easily create a Word document with an extension of .bob, or even no extension, without changing the fact that it is still a Word document or preventing the ability of Word to open and manipulate the file.

A file with an .mdf extension is usually the first data file that is created within a database, generally is associated with the primary filegroup, and usually is considered the primary data file which contains all the system objects necessary to a database. The .ndf extension is generally used for all other data files underneath a database, regardless of the filegroup to which the file is associated. The .ldf extension generally is used for transaction logs.

The file extensions that you see for SQL Server are nothing more than naming conventions. SQL Server does not care what the file extensions are or even if the files have extensions. If you really wanted to, you could use an .ldf extension for the primary data file, just as you could use an .mdf extension for a transaction log file. Although the use of file extensions in this way does not affect SQL Server, it generally could cause confusion among the other database administrators (DBAs) in your organization. To avoid this confusion, it is recommended that you use the .mdf, .ndf, and .ldf naming conventions commonly used across the SQL Server industry, but do not forget that this is just a naming convention and has absolutely no effect on SQL Server itself.

All data manipulation within SQL Server occurs in memory within a set of buffers. If you are adding new data to a database, the new data is first written to a memory buffer, then written to the transaction log, and finally persisted to a data file via a background process called *check pointing*. When you modify or delete an existing row, if the row does not already exist

in memory, SQL Server first reads the data off disk before making the modification. Similarly if you are reading data that has not yet been loaded into a memory buffer, SQL Server must read it out of the data files on disk.

If you could always ensure that the machine hosting your databases had enough memory to hold all the data within your databases, SQL Server could simply read all the data off disk into memory buffers upon startup to improve performance. However, databases are almost always much larger than the memory capacity on any machine, so SQL Server retrieves data from disk only on an as-needed basis. If SQL Server does not have enough room in memory for the data being read in, the least recently used buffer pools are emptied to make room for newly requested data.

Because accessing a disk drive is much slower than accessing memory, the data file design underneath a database can have an impact on performance.

The first layer of design is within the disk subsystem. As the number of disk drives within a volume increases, the read and write throughput for SQL Server increases. However, there is an upper limit on the disk input/output (I/O), which is based upon the capacity of the redundant array of independent disks (RAID) controller, host bus adapter (HBA), and disk bus. So you cannot fix a disk I/O bottleneck by continually adding more disk drives. Although entire 200+ page white papers have been written on random vs. sequential writes, transfer speeds, rotational speeds, calculations of raw disk read/write speeds, and other topics, the process of designing the disk subsystem is reduced to ensuring that you have enough disks along with appropriately sized controllers and disk caches to deliver the read/write throughput required by your database.

If it were simply a matter of the number of disks, there would be far fewer disk I/O bottlenecks in systems. But there is a second layer of data file design: determining how many data files you need and the location of each data file.

SQL Server creates a thread for each file underneath a database. As you increase the number of files underneath a database, SQL Server creates more threads that can be used to read and write data. However, you cannot just create a database with thousands of files to increase its number of threads. This is because each thread consumes memory, taking away space for data to be cached, and even if you could write to all the threads at the same time, you would then saturate the physical disks behind the data files. In addition, managing thousands of data files underneath a database is extremely cumbersome, and if a large percentage of the files need to expand at the same time, you could create enough activity to halt the flow of data within the database.

Due to the competing factors and the simple fact that in the real world, few DBAs have the time to spend running complex byte transfer rate calculations or even to design the disk layer based on a precise knowledge of the data throughput required, designing the data layer is an iterative approach.

Designing the data layer of a database begins with the database creation. When you create a database, it should have three files and two filegroups. You should have a file with an .mdf extension within a filegroup named *PRIMARY*, a file with an .ndf extension in a filegroup with any name that you choose, and the transaction log with an .ldf extension.

NOTE **FILE EXTENSIONS**

As stated in the sidebar "File Extensions," earlier in this chapter, file extensions are nothing more than naming conventions. They do not convey any special capabilities.

Besides being the logical definition for one or more files that defines the storage boundary for an object, filegroups have a property called *DEFAULT*. The purpose of the *DEFAULT* property is to define the filegroup where SQL Server places objects if you do not specify the ON clause during object creation.

When the database is created, the primary filegroup is marked as the default filegroup. After you create the database, you should mark the second filegroup as the default filegroup. By changing the default filegroup, you ensure that any objects you create are not accidentally placed on the primary filegroup and that only the system objects for the database reside on the primary filegroup. You change the default filegroup by using the following command:

```
ALTER DATABASE <database name> MODIFY FILEGROUP <filegroup name> DEFAULT
```

The main reason not to place any of your objects on the primary filegroup is to provide as much isolation in the I/O as possible. The data in the system objects does not change as frequently as data in your objects. By minimizing the write activity to the primary data file, you reduce the possibility of introducing corruption due to hardware failures. In addition, because the state of the primary filegroup also determines the state of the database, you can increase the availability of the database by minimizing the changes made to the primary filegroup.

Following the initial creation of the database, you add filegroups as needed to separate the storage of objects within the database. You also add files to filegroups to increase the disk I/O available to the objects stored on the filegroup, thereby reducing disk bottlenecks.

Transaction Logs

When SQL Server acknowledges that a transaction has been committed, SQL Server must ensure that the change is hardened to persistent storage. Although all writes occur through memory buffers, persistence is guaranteed by requiring that all changes are written to the transaction log prior to a commit being issued. In addition, the writes to the transaction log must occur directly to disk.

Because every change made to a database must be written directly to disk, the disk storage architecture underneath your transaction log is the most important decision affecting the maximum transaction throughput that you can achieve.

SQL Server writes sequentially to the transaction log but does not read from the log except during a restart recovery. Because SQL Server randomly reads and writes to the data files underneath a database, by isolating the transaction log to a dedicated set of disks you ensure that the disk heads do not have to move all over the disk and move in a mostly linear manner.

EXAM TIP

The maximum transaction throughput for any database is bound by the amount of data per second that SQL Server can write to the transaction log.

Benchmarks

Benchmark disclosures are the best source of information when designing the disk storage for optimal performance. Many organizations and the press place great emphasis on various benchmarks. However, a careful study reveals that, by itself, SQL Server doesn't have as large of an impact on the overall numbers as you are led to believe. The transaction processing engine within SQL Server is extremely efficient and has a fixed contribution to transaction throughput, but the real key to maximizing the transaction rate is in the disk storage. Given the same disk configuration, a 7,200 RPM drive delivers about 50 percent of the SQL Server transaction rate of a 15,000 RPM drive. Having 100 disks underneath a transaction log generally doubles the transaction rate of having only 50 disks. In addition, one of the tricks used in benchmarks is to partition a disk such that all the SQL Server data is written to the outside half or less of the disk platter, because based on physics, as the read/write head of a disk moves toward the edge of a circular object, the velocity increases, thereby spinning a larger segment of the disk platter underneath the drive head per unit of time.

FILESTREAM data

Although the volume of data within organizations has been exploding, leading the way in this data explosion is unstructured data. To tackle the problem of storing, managing, and combining the large volumes of unstructured databases with the structured data in your databases, SQL Server 2008 introduced *FILESTREAM*.

The *FILESTREAM* feature allows you to associate files with a database. The files are stored in a folder on the operating system, but are linked directly into a database where the files can be backed up, restored, full-text-indexed, and combined with other structured data.

Although the details of *FILESTREAM* are covered in more detail in Chapter 3, "Tables," and Chapter 5, "Full Text Indexing," to store *FILESTREAM* data within a database, you need to specify where the data will be stored. You define the location for *FILESTREAM* data in a database by designating a filegroup within the database to be used for storage with the *CONTAINS FILESTREAM* property. The *FILENAME* property defined for a *FILESTREAM* filegroup specifies the path to a folder. The initial part of the folder path definition must exist; however, the last folder in the path defined cannot exist and is created automatically. After the *FILESTREAM* folder has been created, a filestream.hdr file is created in the folder, which is a system file used to manage the files subsequently written to the folder.

tempdb Database

Because the *tempdb* database is much more heavily used than in previous versions, special care needs to be taken in how you design the storage underneath *tempdb*.

In addition to temporary objects, SQL Server uses *tempdb* for worktables used in grouping/sorting operations, worktables to support cursors, the version store supporting snapshot isolation level, and overflow for table variables. You can also cause index build operations to use space in *tempdb*.

Due to the potential for heavy write activity, you should move *tempdb* to a set of disks separated from your databases and any backup files. To spread out the disk I/O, you might consider adding additional files to *tempdb*.

> ***NOTE*** **MULTIPLE *tempdb* FILES**
>
> **A common practice for *tempdb* is to create one file per processor. The one file per processor is with respect to what SQL Server would consider a processor and not the physical processor, which could have multiple cores as well as hyperthreading.**

Quick Check

1. **What are the types of files that you create for databases and what are the commonly used file extensions?**
2. **What is the purpose of the transaction log?**

Quick Check Answers

1. **You can create data and log files for a database. Data files commonly have either an .mdf or .ndf extension, whereas log files have an .ldf extension.**
2. **The transaction log records every change that occurs within a database to persist all transactions to disk.**

Creating Databases

In this practice, you create a database with multiple files that is enabled for *FILESTREAM* storage.

1. Execute the following code to create a database:

```
CREATE DATABASE TK432 ON  PRIMARY
( NAME = N'TK432_Data', FILENAME = N'c:\test\TK432.mdf' ,
    SIZE = 8MB , MAXSIZE = UNLIMITED, FILEGROWTH = 16MB ),
 FILEGROUP FG1
( NAME = N'TK432_Data2', FILENAME = N'c:\test\TK432.ndf' ,
    SIZE = 8MB , MAXSIZE = UNLIMITED, FILEGROWTH = 16MB ),
```

```
 FILEGROUP Documents CONTAINS FILESTREAM  DEFAULT
( NAME = N'Documents', FILENAME = N'c:\test\TK432Documents' )
 LOG ON
( NAME = N'TK432_Log', FILENAME = N'c:\test\TK432.ldf' ,
    SIZE = 8MB , MAXSIZE = 2048GB , FILEGROWTH = 16MB )
GO
```

2. Execute the following code to change the default filegroup:

```
ALTER DATABASE TK432
MODIFY FILEGROUP FG1
DEFAULT
GO
```

Lesson Summary

- You can define one or more data and log files for the physical storage of a database.
- Data files are associated to a filegroup within a database.
- Filegroups provide the logical storage container for objects within a database.
- Files can be stored using the new *FILESTREAM* capabilities.

Lesson Review

The following question is intended to reinforce key information presented in Lesson 1, "Configuring Files and Filegroups." The question is also available on the companion CD if you prefer to review it in electronic form.

> ***NOTE*** **ANSWERS**
>
> **Answers to this question and an explanation of why each answer choice is correct or incorrect is located in the "Answers" section at the end of the book.**

1. You have a reference database named *OrderHistory,* which should not allow any data to be modified. How can you ensure, with the least amount of effort, that users can only read data from the database?
   - **A.** Add all database users to the db_datareader role.
   - **B.** Create views for all the tables and grant select permission only on the views to database users.
   - **C.** Set the database to READ_ONLY.
   - **D.** Grant select permission on the database to all users and revoke insert, update, and delete permissions from all users on the database.

Lesson 2: Configuring Database Options

Data within a database is stored on disk in one or more data files. Prior to being written to the data file(s), every transaction is written to a transaction log file. In this lesson, you learn how to design the data files underneath a database, group the files into filegroups to link physical storage into a database, and manage the transaction log.

After this lesson, you will be able to:

- Set the database recovery model
- Configure database options
- Manage collation sequences
- Check and maintain database consistency

Estimated lesson time: 20 minutes

Database Options

A database has numerous options that control a variety of behaviors. These options are broken down into several categories, including the following:

- Recovery
- Auto options
- Change tracking
- Access
- Parameterization

Recovery Options

The recovery options determine the behavior of the transaction log and how damaged pages are handled.

Recovery Models

Every database within a SQL Server instance has a property setting called the *recovery model*. The recovery model determines the types of backups you can perform against a database. The recovery models available in SQL Server 2008 are:

- Full
- Bulk-logged
- Simple

THE FULL RECOVERY MODEL

When a database is in the Full recovery model, all changes made, using both data manipulation language (DML) and data definition language (DDL), are logged to the transaction log. Because all changes are recorded in the transaction log, it is possible to recover a database in the Full recovery model to a given point in time so that data loss can be minimized or eliminated if you should need to recover from a disaster. Changes are retained in the transaction log indefinitely and are removed only by executing a transaction log backup.

> ***BEST PRACTICES*** **RECOVERY MODELS**
>
> **Every production database that accepts transactions should be set to the Full recovery model. By placing the database in the Full recovery model, you can maximize the restore options that are possible.**

THE BULK-LOGGED RECOVERY MODEL

Certain operations are designed to manipulate large amounts of data. However, the overhead of logging to the transaction log can have a detrimental impact on performance. The Bulk-logged recovery model allows certain operations to be executed with minimal logging. When a minimally logged operation is performed, SQL Server does not log every row changed but instead logs only the extents, thereby reducing the overhead and improving performance. The operations that are performed in a minimally logged manner with the database set in the Bulk-logged recovery model are:

- *BCP*
- *BULK INSERT*
- *SELECT…INTO*
- *CREATE INDEX*
- *ALTER INDEX…REBUILD*

Because the Bulk-logged recovery model does not log every change to the transaction log, you cannot recover a database to a point in time, within the interval that a minimally logged transaction executed, when the Bulk-logged recovery model was enabled.

THE SIMPLE RECOVERY MODEL

The third recovery model is Simple. A database in the Simple recovery model logs operations to the transaction log exactly as the Full recovery model does. However, each time the database checkpoint process executes, the committed portion of the transaction log is discarded. A database in the Simple recovery model cannot be recovered to a point in time because it is not possible to issue a transaction log backup for a database in the simple recovery model.

Because the recovery model is a property of a database, you set the recovery model by using the *ALTER DATABASE* command as follows:

```
ALTER DATABASE database_name
SET RECOVERY { FULL | BULK_LOGGED | SIMPLE }
```

The backup types available for each recovery model are shown in Table 2-1.

TABLE 2-1 Backup Types Available for Each Recovery Model

RECOVERY MODEL	BACKUP TYPE		
	FULL	**DIFFERENTIAL**	**TRAN LOG**
Full	Yes	Yes	Yes
Bulk	Yes	Yes	Yes/no minimally logged
Simple	Yes	Yes	No

EXAM TIP

You need to know which types of backups are possible for each recovery model.

Damaged Pages

It is possible to damage data pages during a write to disk if you have a power failure or failures in disk subsystem components during the write operation. If the write operation fails to complete, you can have an incomplete page in the database that cannot be read. Because the damage happens to a page on disk, the only time that you see a result of the damage is when SQL Server attempts to read the page off disk.

The default configuration of SQL Server does not check for damaged pages and could cause the database to go off-line if a damaged page is encountered. The PAGE_VERIFY CHECKSUM option can be enabled, which allows you to discover and log damaged pages. When pages are written to disk, a checksum for the page is calculated and stored in the page header. When SQL Server reads a page from disk, a checksum is calculated and compared to the checksum stored in the page header. If a damaged page is encountered, an 824 error is returned to the calling application and logged to the SQL Server error log and Windows Application Event log, and the ID of the damaged page is logged to the suspect_pages table in the *msdb* database.

In SQL Server 2005, the only way to fix a damaged page was to execute a page restore, which is discussed in Chapter 9, "Backing Up and Restoring a Database." In addition to a page restore, if the database is participating in a database mirroring session, SQL Server 2008 automatically replaces the page with a copy of the page from the mirror. When Database Mirroring automatically fixes a corrupt page, an entry is logged and can be reviewed with the sys.dm_db_mirroring_auto_page_repair view.

Auto Options

There are five options for a database that enable certain actions to occur automatically:

- AUTO_CLOSE
- AUTO_SHRINK

- AUTO_CREATE_STATISTICS
- AUTO_UPDATE_STATISTICS
- AUTO_UPDATE_STATISTICS_ASYNCH

Each database within an instance requires a variety of resources, the most significant of which is a set of memory buffers. Each open database requires several bytes of memory and any queries against the database populate the data and query caches. If the AUTO_CLOSE option is enabled, when the last connection to a database is closed, SQL Server shuts down the database and releases all resources related to the database. When a new connection is made to the database, SQL Server starts up the database and begins allocating resources.

By default, AUTO_CLOSE is disabled. Unless you have severe memory pressure, you should not enable a database for AUTO_CLOSE. In addition, a database that is frequently accessed should not be set to AUTO_CLOSE because it would cause a severe degradation in performance. This is because you would never be able to use the data and query caches adequately.

Data files can be set to grow automatically when additional space is needed. Although most operations to increase space affect the database on a long-term basis, some space increases are needed only on a temporary basis. If the AUTO_SHRINK option is enabled, SQL Server periodically checks the space utilization of data and transaction log files. If the space checking algorithm finds a data file that has more that 25 percent free space, the file automatically shrinks to reclaim disk space.

Expanding a database file is a very expensive operation. Shrinking a database file is also an expensive operation. If the size of a database file increased during normal operations, it is very likely that if the file shrinks, the operation would recur and increase the database file again. The only operations that cause one-time space utilization changes to database files are administrative processes that create and rebuild indexes, archive data, or load data. Because the growth of database files is so expensive, it is recommended to leave the AUTO_SHRINK option disabled and manually shrink files only when necessary.

Statistics allow the Query Optimizer to build more efficient query plans. If the AUTO_CREATE_STATSTICS option is enabled, SQL Server automatically creates statistics that are missing during the optimization phase of query processing. Although the creation of statistics incurs some overhead, the benefit to query performance is worth the overhead cost for SQL Server to create statistics automatically when necessary.

Statistics capture the relative distribution of values in one or more columns of a table. After the database has been in production for a while, normal database changes do not appreciably change the statistics distribution in general. However, mass changes to the data or dramatic shifts in business processes can suddenly introduce significant skew into the data. If the statistics are not updated to reflect the distribution shift, the Optimizer could select an inefficient query plan.

Databases have two options that allow SQL Server to update out-of-date statistics automatically. The AUTO_UPDATE_STATISTICS option updates out-of-date statistics during query optimization. If you choose to enable AUTO_UPDATE_STATISTICS, a second

option, AUTO_UPDATE_STATISTICS_ASYNC, controls whether statistics are updated during query optimization or if query optimization continues while the statistics are updated asynchronously.

Change Tracking

One of the challenges for any multiuser system is to ensure that the changes of one user do not accidentally overwrite the changes of another. To prevent the changes of multiple users from overriding each other, applications are usually built within mechanisms to determine whether a row has changed between the time it was read and the time it is written back to the database. The tracking mechanisms usually involve columns with either a datetime or timestamp column and also might include an entire versioning system.

SQL Server 2008 introduces a new feature implemented through the CHANGE_TRACKING database option. Change tracking is a lightweight mechanism that associates a version with each row in a table that has been enabled for change tracking. Each time the row is changed, the version number is incremented. Instead of building systems to avoid changes from multiple users overriding each other, applications need only compare the row version to determine if a change has occurred to the row between when the row was read and written.

After change tracking has been enabled for the database, you can choose which tables within a database that change tracking information should be captured for. Over time, change tracking information accumulates in the database, so you can also specify how long tracking information is retained through the CHANGE_RETENTION option and whether tracking information should be automatically cleaned up with the AUTO_CLEANUP option.

Access

Access to a database can be controlled through several options.

The status of a database can be explicitly set to ONLINE, OFFLINE, or EMERGENCY. When a database is in an ONLINE state, you can perform all operations that would otherwise be possible. A database that is in an OFFLINE state is inaccessible. A database in an EMERGENCY state can be accessed only by a member of the db_owner role, and the only command allowed to be executed is *SELECT*.

You can control the ability to modify data for an online database by setting the database to either READ_ONLY or READ_WRITE. A database in READ_ONLY mode cannot be written to. In addition, when a database is placed in READ_ONLY mode, SQL Server removes any transaction log file that is specified for the database. Changing a database from READ_ONLY to READ_WRITE causes SQL Server to re-create the transaction log file.

User access to a database can be controlled through the SINGLE_USER, RESTRICTED_USER, and MULTI_USER options. When a database is in SINGLE_USER mode, only a single user is allowed to access the database. A database set to RESTRICTED_USER only allows access to members of the db_owner, dbcreator, and sysadmin roles.

If multiple users are using the database when you change the mode to SINGLE_USER or users that conflict with the allowed set for RESTRICTED_USER, the *ALTER DATABASE* command is blocked until all the non-allowed users disconnect. Instead of waiting for users to complete operations and disconnect from the database, you can specify a ROLLBACK action to terminate connections forcibly. The ROLLBACK IMMEDIATE option forcibly rolls back any open transactions, along with disconnecting any nonallowed users. You can allow users to complete transactions and exit the database by using the ROLLBACK AFTER <*number of seconds*> option, which waits for the specified number of seconds before rolling back transactions and disconnecting users.

The normal operational mode for most databases is ONLINE, READ_WRITE, and MULTI_USER.

Parameterization

One of the "hot button" topics in application development is whether to parameterize calls to the database. When a database call is parameterized, the values are passed as variables. You can find just as many articles advocating for both sides. Unfortunately, applications gain a significant benefit when database calls are parameterized.

SQL Server caches the query plan for every query that is executed. Unless there is pressure on the query cache that forces a query plan from the cache, every query executed since the instance started is in the query cache. When a query is executed, SQL Server parses and compiles the query. The query is then compared to the query cache using a string-matching algorithm. If a match is found, SQL Server retrieves the plan that has already been generated and executes the query.

A query that is parameterized has a much higher probability of being matched because the query string does not change even when the values being used vary. Therefore, parameterized queries can reuse cached query plans more frequently and avoid the time required to build a query plan.

Because not all applications parameterize calls to the database, you can force SQL Server to parameterize every query for a given database by setting the PARAMETERIZATION FORCED database option.

The default setting for a database is not to force parameterization. The reuse of query plans provides a benefit so long as the query plan being reused is the most efficient path through the data. For tables where there is significant data skew, one value produces an efficient query plan, whereas another value causes a different query plan to be created. In addition, applications see the effect of parameterization only if the majority of database calls have an extremely short duration.

So long as the majority of your database calls have a very short duration and the query plan generated do not change depending upon the parameters passed, you could see a performance boost by forcing parameterization.

Collation Sequences

SQL Server has the capability to store character data that spans every possible written language. However, not every language follows the same rules for sorting or data comparisons. SQL Server allows you to define the rules for comparison, sorting, case sensitivity, and accent sensitivity through the specification of a collation sequence.

When you install SQL Server, you specify a default collation sequence that is used for all databases, tables, and columns. You can override the default collation sequence at each level. The collation sequence for an instance can be overridden at a database level by specifying the COLLATE clause in either the *CREATE DATABASE* or *ALTER DATABASE* command.

Quick Check

1. How do you restrict database access to members of the db_owner role and terminate all active transactions and connection at the same time?
2. What backups can be executed for a database in each of the recovery models?

Quick Check Answers

1. You would execute the following command: ALTER DATABASE <*database name*> SET RESTRICTED_USER WITH ROLLBACK IMMEDIATE.
2. You can create full, differential, and file/filegroup backups in the Simple recovery model. The Bulk-logged recovery model allows you to execute types of backups, but you cannot restore a database to a point in time during an interval when a minimally logged transaction is executing. All types of backups can be executed in the Full recovery model.

Changing the Database Recovery Model

In this practice, you change the recovery model of the *AdventureWorks* database to *FULL* to ensure that you can recover from a failure to a point in time.

1. Execute the following code:

```
ALTER DATABASE AdventureWorks
    SET RECOVERY FULL
GO
```

2. Right-click the *AdventureWorks* database, select Properties, and select the Options tab to view the recovery model and make sure that it is full.

Lesson Summary

- You can set the recovery model for a database to Full, Bulk-logged, or Simple.
- You can back up transaction logs for a database in the Full or Bulk-logged recovery model.
- The AUTO_SHRINK option shrinks a database file when there is more than 25 percent of free space in the file.
- You can track and log damaged pages by enabling the PAGE_VERIFY CHECKSUM option.

Lesson Review

The following question is intended to reinforce key information presented in Lesson 2, "Configuring Database Options." The question is also available on the companion CD if you prefer to review it in electronic form.

> ***NOTE*** **ANSWERS**
>
> **Answers to this question and an explanation of why each answer choice is correct or incorrect is located in the "Answers" section at the end of the book.**

1. You are the database administrator at Blue Yonder Airlines and are primarily responsible for the *Reservations* database, which runs on a server running SQL Server 2008. In addition to customers booking flights through the company's Web site, flights can be booked with several partners. Once an hour, the *Reservations* database receives multiple files from partners, which are then loaded into the database using the Bulk Copy Program (BCP) utility. You need to ensure that you can recover the database to any point in time while also maximizing the performance of import routines. How would you configure the database to meet business requirements?

   A. Enable AUTO_SHRINK

   B. Set PARAMETERIZATION FORCED on the database

   C. Configure the database in the Bulk-logged recovery model

   D. Configure the database in the Full recovery model

Lesson 3: Maintaining Database Integrity

In a perfect world, everything that you save to disk storage would always write correctly, read correctly, and never have any problems. Unfortunately, your SQL Server databases live in an imperfect world where things do go wrong. Although this occurs very rarely, data within your database can become corrupted if there is a failure in the disk storage system as SQL Server is writing to a page. Data pages are 8 kilobytes (KB) in size, but SQL Server divides a page into 16 blocks of 512 bytes apiece when performing write operations. If SQL Server begins writing blocks on a page and the disk system fails in the middle of the write process, only a portion of the page is written successfully, producing a problem called a *torn page*. In this lesson, you learn how to detect and correct corruption errors in your database.

After this lesson, you will be able to:

- Check a database for integrity
- Use DMVs to diagnose corruption issues

Estimated lesson time: 20 minutes

Database Integrity Checks

As you learned in Lesson 2, databases have an option called *PAGE_VERIFY*. The page verification can be set to either *TORN_PAGE_DETECTION* or *CHECKSUM*. The *PAGE_VERIFY TORN_PAGE_DETECTION* option exists for backwards compatibility and should not be used. When the *PAGE_VERIFY CHECKSUM* option is enabled, SQL Server calculates a checksum for the page prior to the write. Each time a page is read off disk, a checksum is recalculated and compared to the checksum written to the page. If the checksums do not match, the page has been corrupted.

When SQL Server encounters a corrupt page, an error is thrown, the command attempting to access the corrupt page is aborted, and an entry is written into the suspect_pages table in the *msdb* database.

BEST PRACTICES **PAGE VERIFICATION**

You should enable the *PAGE_VERIFY CHECKSUM* option on every production database.

Although page verification can detect and log corrupted pages, the page must be read off disk to trigger the verification check. Data is normally read off disk when users and applications access data, but instead of having a user receive an error message, it is much better for you to proactively find corruption and fix the problem by using a backup before the user has a process aborted.

You can force SQL Server to read every page from disk and check the integrity by executing the *DBCC CHECKDB* command. The generic syntax of *DBCC CHECKDB* is:

```
DBCC CHECKDB [( 'database_name' | database_id | 0
    [ , NOINDEX | { REPAIR_ALLOW_DATA_LOSS | REPAIR_FAST
    | REPAIR_REBUILD } ] )]
    [ WITH {[ ALL_ERRORMSGS ] [ , [ NO_INFOMSGS ] ] [ , [ TABLOCK ] ]
           [ , [ ESTIMATEONLY ] ] [ , [ PHYSICAL_ONLY ] ] | [ , [ DATA_PURITY ] ] } ]
```

When *DBCC CHECKDB* is executed, SQL Server performs all the following actions:

- Checks page allocation within the database
- Checks the structural integrity of all tables and indexed views
- Calculates a checksum for every data and index page to compare against the stored checksum
- Validates the contents of every indexed view
- Checks the database catalog
- Validates Service Broker data within the database

To accomplish these checks, *DBCC CHECKDB* executes the following commands:

- *DBCC CHECKALLOC*, to check the page allocation of the database
- *DBCC CHECKCATALOG*, to check the database catalog
- *DBCC CHECKTABLE*, for each table and view in the database to check the structural integrity

Any errors encountered are output so that you can fix the problems. If an integrity error is found in an index, you should drop and re-create the index. If an integrity error is found in a table, you need to use your most recent backups to repair the damaged pages.

NOTE **DATABASE MIRRORING**

If the database is participating in Database Mirroring, SQL Server attempts to retrieve a copy of the page from the mirror. If the page can be retrieved from the mirror and has the correct page contents, the page is replaced automatically on the principal without requiring any intervention. When SQL Server replaces a corrupt page from the mirror, an entry is written into the *sys.dm_db_mirroring_auto_page_repair* view.

Quick Check

1. **Which option should be enabled for all production databases?**
2. **What checks does *DBCC CHECKDB* perform?**

Quick Check Answers

1. **You should set the *PAGE_VERIFY CHECKSUM* option for all production databases.**
2. ***DBCC CHECKDB* checks the logical and physical integrity of every table, index, and indexed view within the database, along with the contents of every indexed view, page allocations, Service Broker data, and database catalog.**

Checking Database Integrity

In this practice, you check the integrity of the *AdventureWorks* database.

1. Execute the following code:

```
DBCC CHECKDB ('AdventureWorks') WITH NO_INFOMSGS, ALL_ERRORMSGS
GO
```

2. Review the results.

Lesson Summary

- The *PAGE_VERIFY CHECKSUM* option should be enabled for every production database to detect any structural integrity errors.
- When a corrupt page is encountered, the page is logged to the suspect_pages table in the *msdb* database. If a database is participating in a Database Mirroring session, SQL Server automatically retrieves a copy of the page from the mirror, replaces the page on the principal, and logs an entry in the *sys.dm_db_mirroring_auto_page_repair* view.
- *DBCC CHECKDB* is used to check the logical and physical consistency of a database.

Lesson Review

The following question is intended to reinforce key information presented in Lesson 3, "Maintaining Database Integrity." The question is also available on the companion CD if you prefer to review it in electronic form.

> ***NOTE*** **ANSWERS**
>
> **Answers to this question and an explanation of why each answer choice is correct or incorrect is located in the "Answers" section at the end of the book.**

1. Which commands are executed when you run the *DBCC CHECKDB* command? (Check all that apply.)
   - **A.** *DBCC CHECKTABLE*
   - **B.** *DBCC CHECKIDENT*
   - **C.** *DBCC CHECKCATALOG*
   - **D.** *DBCC FREEPROCCACHE*

Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can perform the following tasks:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenario. This scenario sets up a real-world situation involving the topics in this chapter and asks you to create a solution.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- Databases can be configured with the Full, Bulk-logged, or Simple recovery model.
- The recovery model of the database determines the backups that can be created, as well as limitations on the recovery options that can be performed.
- You can set a collation sequence for a database that overrides the collation sequence defined for the instance.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Corrupt page
- Filegroup
- Recovery model

Case Scenario

In the following case scenario, you apply what you've learned in this chapter. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario: Configuring Databases for Coho Vineyard

BACKGROUND

Company Overview

Coho Vineyard was founded in 1947 as a local, family-run winery. Due to the award-winning wines it has produced over the last several decades, Coho Vineyards has experienced significant growth. To continue expanding, several existing wineries were acquired over the years. Today, the company owns 16 wineries; 9 wineries are in Washington, Oregon, and California, and the remaining 7 wineries are located in Wisconsin and Michigan. The wineries

employ 532 people, 162 of whom work in the central office that houses servers critical to the business. The company has 122 salespeople who travel around the world and need access to up-to-date inventory availability.

Planned Changes

Until now, each of the 16 wineries owned by Coho Vineyard has run a separate Web site locally on the premises. Coho Vineyard wants to consolidate the Web presence of these wineries so that Web visitors can purchase products from all 16 wineries from a single online store. All data associated with this Web site be stored in databases in the central office.

When the data is consolidated at the central office, merge replication will be used to deliver data to the salespeople as well as to allow them to enter orders. To meet the needs of the salespeople until the consolidation project is completed, inventory data at each winery is sent to the central office at the end of each day. Merge replication has been implemented to allow salespeople to maintain local copies of customer, inventory, and order data.

EXISTING DATA ENVIRONMENT

Databases

Each winery presently maintains its own database to store all business information. At the end of each month, this information is brought to the central office and transferred into the databases shown in Table 2-2.

TABLE 2-2 Coho Vineyard Databases

DATABASE	SIZE
Customer	180 megabytes (MB)
Accounting	500 MB
HR	100 MB
Inventory	250 MB
Promotions	80 MB

After the database consolidation project is complete, a new database named *Order* will serve as a data store to the new Web store. As part of their daily work, employees also will connect periodically to the *Order* database using a new in-house Web application.

The *HR* database contains sensitive data and is protected using Transparent Data Encryption (TDE). In addition, data in the Salary table is encrypted using a certificate.

Database Servers

A single server named DB1 contains all the databases at the central office. DB1 is running SQL Server 2008 Enterprise on Windows Server 2003 Enterprise.

Business Requirements

You need to design an archiving solution for the *Customer* and *Order* databases. Your archival strategy should allow the Customer data to be saved for six years.

To prepare the *Order* database for archiving procedures, you create a partitioned table named Order.Sales. Order.Sales includes two partitions. Partition 1 includes sales activity for the current month. Partition 2 is used to store sales activity for the previous month. Orders placed before the previous month should be moved to another partitioned table named Order.Archive. Partition 1 of Order.Archive includes all archived data. Partition 2 remains empty.

A process needs to be created to load the inventory data from each of the 16 wineries by 4 A.M. daily.

Four large customers submit orders using Coho Vineyards Extensible Markup Language (XML) schema for Electronic Data Interchange (EDI) transactions. The EDI files arrive by 5 P.M. and need to be parsed and loaded into the *Customer, Accounting,* and *Inventory* databases, which each contain tables relevant to placing an order. The EDI import routine is currently a single-threaded C++ application that takes between three and six hours to process the files. You need to finish the EDI process by 5:30 P.M. to meet your Service Level Agreement (SLA) with the customers. After the consolidation project has finished, the EDI routine loads all data into the new *Order* database.

You need to back up all databases at all locations. You can lose a maximum of five minutes of data under a worst-case scenario. The *Customer, Account, Inventory, Promotions,* and *Order* databases can be off-line for a maximum of 20 minutes in the event of a disaster. Data older than six months in the *Customer* and *Order* databases can be off-line for up to 12 hours in the event of a disaster.

Answer the following questions.

1. How should you configure the databases for maximum performance?
2. How should the databases be configured to meet recovery obligations?

Suggested Practices

To help you master the exam objectives presented in this chapter, complete the following tasks.

Configuring Databases

- **Practice 1** Create a database which can store *FILESTREAM* data.
- **Practice 2** Change the recovery model and observe the effects on backup and restore options.

- **Practice 3** Change the database state to READ_ONLY and observe the effect on the transaction log file.
- **Practice 4** Create multiple connections to a database, change the access to RESTRICTED_USER, and specify the ROLLBACK IMMEDIATE option. Observe the effects.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just one exam objective, or you can test yourself on all the 70-432 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO **PRACTICE TESTS**

For details about all the practice test options available, see the section entitled "How to Use the Practice Tests," in the Introduction to this book.